**Documentazione su River**

**Data streams**

River è una libreria per costruire modelli di online machine learning. Tali modelli operano su *data streams*. Un data stream è un concetto un po’ vago.

In generale, un *data stream* è una sequenza di elementi individuali. Nel caso del machine learning, ogni elemento è un insieme di *features*. Chiamiamo questi elementi *samples* o *observations*. Ogni sample può seguire una struttura fissa e contenere sempre le stesse features. Ma le features possono anche apparire e scomparire nel tempo. Questo dipende dal caso d’uso.

**Reactive e proactive data streams**

L’origine di un data stream può variare, e di solito non importa. Dovresti essere in grado di usare River indipendentemente da dove provengano i tuoi dati. È però importante tenere a mente la differenza tra *reactive* e *proactive**data streams*.

* I *reactive* data streams sono dati che arrivano a te. Ad esempio, quando un utente visita il tuo sito web, questo è fuori dal tuo controllo. Non hai influenza sull’evento. Succede e basta, e tu devi reagire.
* I *proactive* data streams sono dati di cui puoi controllarne il flusso. Ad esempio, potresti leggere i dati da un file. Decidi a quale velocità vuoi leggere i dati, in quale ordine, ecc.

Se consideri l’analisi dei dati nel suo insieme, ti rendi conto che l’approccio generale è trasformare reactive streams in proactive datasets. Gli eventi vengono di solito loggati in un database e processati offline. Che sia per costruire KPI o per addestrare modelli.

La sfida per il machine learning è garantire che i modelli addestrati offline su proactive datasets funzionino correttamente in produzione su reactive data streams.

**Online processing**

L’*online processing* è l’atto di processare un data stream un elemento alla volta. Nel caso del machine learning, significa addestrare un modello insegnandogli un sample alla volta. Questo è completamente opposto al metodo tradizionale di fare machine learning, che consiste nell’addestrare un modello su interi batch di dati contemporaneamente.

Un modello online è quindi un oggetto *stateful* e dinamico. Continua ad apprendere e non deve rivedere i dati passati. È un modo diverso di fare le cose, e quindi ha il suo set di pro e contro.

**Tasks**

Il machine learning comprende molti task diversi: classificazione, regressione, anomaly detection, time series forecasting, ecc. L’ideologia dietro *River* è quello di fornire un approccio al machine learning generico che permetta di eseguire questi task in modalità streaming. Infatti, molti algoritmi di machine learning classici (*batch*) hanno equivalenti online.

Nota che River supporta anche task più semplici. Ad esempio, potresti voler semplicemente calcolare una media mobile di un data stream. Questi sono solitamente componenti più piccoli di una pipeline di stream processing completa.

**Dictionaries everywhere**

River è una libreria Python. È composta da una serie di classi che implementano vari algoritmi di online processing. La maggior parte di queste classi sono modelli di machine learning che possono processare un singolo sample, sia per apprendimento che per inferenza.

Abbiamo scelto di usare i *dictionaries* come building block di base. Prima di tutto, l’online processing è diverso dal batch processing, e di conseguenza la vettorizzazione non porta nessun aumento di velocità. Quindi librerie numeriche come NumPy e PyTorch in realtà aggiungono solo overhead. Usare le strutture dati native di Python è più veloce.

I dictionaries sono quindi perfetti: sono nativi in Python e hanno un eccellente supporto nella standard library. Permettono di nominare ogni feature. Possono contenere qualsiasi tipo di dato. Consentono un supporto trasparente dei payload JSON, permettendo un’integrazione fluida con le web app.

**Datasets**

In produzione, quasi sempre ti troverai davanti a data streams a cui devi reagire, come gli utenti che visitano il tuo sito. Il vantaggio dell’online machine learning è che puoi progettare modelli che fanno previsioni e allo stesso tempo imparano da questo data stream man mano che scorre.

Ma ovviamente, quando sviluppi un modello, di solito non hai accesso a un feed in tempo reale su cui valutarlo. Di solito hai un dataset offline con cui vuoi testare il tuo modello. River fornisce alcuni datasets che possono essere letti in modalità online, un sample alla volta. È però cruciale tenere a mente che l’obiettivo è riprodurre uno scenario di produzione il più fedelmente possibile, così da garantire che il tuo modello funzioni altrettanto bene in produzione.

**Model evaluation**

La valutazione di un modello online è diversa dalla sua controparte batch tradizionale. Nel secondo caso, di solito si usa la cross-validation, in cui il dataset viene diviso in training ed evaluation set. Questo va bene, ma non riflette esattamente il processo di generazione dei dati che avviene in produzione.

La valutazione di un modello online implica apprendimento e inferenza nello stesso ordine in cui accadrebbero in produzione. Infatti, se conosci l’ordine in cui i tuoi dati arrivano, puoi processarli nello stesso ordine esatto. Questo ti permette di replicare uno scenario di produzione e valutare il tuo modello con una fedeltà maggiore rispetto alla cross-validation.

Ed è proprio questo che rende potente l’online machine learning. Riproducendo i datasets nel giusto ordine, ti assicuri di progettare modelli che si comporteranno come previsto in produzione.

**Concept drift**

La ragione principale per cui un modello offline potrebbe non funzionare come previsto in produzione è il *concept drift*. Ma questo vale per tutti i modelli di machine learning, siano essi offline o online.

Il vantaggio dei modelli online rispetto a quelli offline è che possono gestire il drift. Infatti, poiché continuano ad apprendere, di solito si adattano al concept drift in modo fluido. Al contrario, i modelli batch devono essere riaddestrati da zero.

**Classificazione binaria**

La classificazione riguarda la previsione di un outcome da una lista fissa di classi. La previsione è una distribuzione di probabilità che assegna una probabilità a ciascun possibile outcome.

Un sample etichettato per classificazione è composto da un insieme di features e da una classe. La classe è booleana nel caso di binary classification. Usiamo come esempio il dataset phishing.

from river import datasets

dataset = datasets.Phishing()

dataset

**Phishing websites**.

Questo dataset contiene features prese da pagine web classificate come phishing o meno.

Name Phishing

Task Binary classification

Samples 1,250

Features 9

Sparse False

Path /home/runner/work/river/river/river/datasets/phishing.csv.gz

Questo dataset è un *streaming dataset*, che può essere iterato.

for x, y in dataset:

pass

Diamo un’occhiata al primo sample:

x, y = next(iter(dataset))

x

{

'empty\_server\_form\_handler': 0.0,

'popup\_window': 0.0,

'https': 0.0,

'request\_from\_other\_domain': 0.0,

'anchor\_from\_other\_domain': 0.0,

'is\_popular': 0.5,

'long\_url': 1.0,

'age\_of\_domain': 1,

'ip\_in\_url': 1

}

y

True

L’obiettivo di un classificatore binario è imparare a predire un target binario Y a partire da features X. Proviamo a farlo con una **logistic regression**.

from river import linear\_model

model = linear\_model.LogisticRegression()

model.predict\_proba\_one(x)

{False: 0.5, True: 0.5}

Il modello non è stato ancora addestrato, quindi restituisce una probabilità di default del 50% per ciascuna classe.

Possiamo addestrarlo sul sample, aggiornando lo stato del modello:

model.learn\_one(x, y)

Ora se facciamo una previsione sullo stesso sample, le probabilità cambiano, perché il modello ha appreso qualcosa:

model.predict\_proba\_one(x)

{False: 0.494687699901455, True: 0.505312300098545}

Nota che esiste anche predict\_one se sei interessato solo alla classe più probabile invece che alla distribuzione di probabilità:

model.predict\_one(x)

True

**Progressive validation**

Tipicamente, un modello online fa una previsione e poi apprende una volta che il ground truth (vera etichetta) si rivela. La previsione e il ground truth possono essere confrontati per misurare la correttezza del modello.

Se hai un dataset disponibile, puoi iterarlo, fare una previsione, aggiornare il modello e confrontare l’output con il ground truth. Questo si chiama *progressive validation*.

from river import metrics

model = linear\_model.LogisticRegression()

metric = metrics.ROCAUC()

for x, y in dataset:

y\_pred = model.predict\_proba\_one(x)

model.learn\_one(x, y)

metric.update(y, y\_pred)

metric

ROCAUC: 89.36%

Questo è un metodo comune per valutare un modello online. In realtà esiste anche una funzione dedicata: evaluate.progressive\_val\_score.

from river import evaluate

model = linear\_model.LogisticRegression()

metric = metrics.ROCAUC()

evaluate.progressive\_val\_score(dataset, model, metric)

ROCAUC: 89.36%

**Migliorare la logistic regression con scaling**

Un modo comune per migliorare la performance di una logistic regression è scalare i dati. Questo si può fare usando un preprocessing.StandardScaler.

In particolare, possiamo definire una *pipeline* per organizzare il modello in una sequenza di step:

from river import compose

from river import preprocessing

model = compose.Pipeline(

preprocessing.StandardScaler(),

linear\_model.LogisticRegression()

)

model

Immagine che contiene testo, Carattere, schermata, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

metric = metrics.ROCAUC()

evaluate.progressive\_val\_score(dataset, model, metric)

ROCAUC: 95.07%

**Concept drift**

Nell’online machine learning, si assume che i dati possano cambiare nel tempo. Quando costruiamo modelli di machine learning, assumiamo che i dati abbiano una distribuzione di probabilità che è di solito fissa, cioè stazionaria. I cambiamenti nella distribuzione dei dati danno origine al fenomeno chiamato *Concept drift*. Tali drift possono essere virtuali o reali.

Nei virtual drifts, cambia solo la distribuzione delle features mentre la relazione tra le features e il target rimane invariata. La probabilità congiunta delle variabili cambia nei real concept drifts. Di conseguenza, i problemi di online machine learning non supervisionato potrebbero affrontare solo virtual concept drifts.

I real concept drifts possono essere ulteriormente divisi in *abrupt* (succedono istantaneamente in un punto dato) o *gradual* (un “concept” cambia in un altro gradualmente). Esistono altre possibili suddivisioni, ma possono essere ricondotte a abrupt o gradual drifts.

**Esempi di concept drift**

I concept drifts possono accadere nella domanda di elettricità durante l’anno, nel mercato azionario, nelle preferenze di acquisto e nella probabilità di successo di un nuovo film, tra gli altri.

Consideriamo l’esempio dei film: due film realizzati in epoche diverse possono avere features simili come attori/registi famosi, trama, budget di produzione, campagne di marketing, ecc., e tuttavia non è certo che entrambi avranno lo stesso successo. Ciò che il pubblico target considera degno di essere guardato (e di spendere i propri soldi) cambia costantemente, e le case di produzione devono adattarsi di conseguenza per evitare “box office flops”.

Prima della pandemia, l’uso di disinfettanti per le mani e mascherine non era diffuso. Quando i casi di COVID-19 hanno iniziato ad aumentare, c’è stata una mancanza di tali prodotti per i consumatori finali. Immagina un modello di batch-learning che decide quanta scorta di ciascun prodotto un supermercato dovrebbe tenere in quel periodo. Che disastro!

**L’impatto del drift nell’apprendiemento**

Il concept drift può avere un impatto significativo sulla performance predittiva se non gestito correttamente. La maggior parte dei modelli di batch learning fallirà in presenza di concept drift, poiché sono essenzialmente addestrati su dati diversi. D’altra parte, i metodi di stream learning si aggiornano continuamente e si adattano ai nuovi concetti. Inoltre, i metodi drift-aware usano tecniche di change detection (drift detectors) per attivare meccanismi di mitigazione se viene rilevato un cambiamento nella performance.

**Detecting concept drift**

Sono stati proposti diversi metodi di drift detection. L’obiettivo di un drift detector è segnalare un allarme in presenza di drift. Un buon drift detector massimizza il numero di true positives riducendo al minimo il numero di false positives. Deve anche essere efficiente in termini di risorse per funzionare nel contesto di infinite data streams.

Per questo esempio, genereremo un synthetic data stream concatenando 3 distribuzioni di 1000 samples ciascuna.

import numpy as np

import matplotlib.pyplot as plt

from matplotlib import gridspec

# Genera dati per 3 distribuzioni

random\_state = np.random.RandomState(seed=42)

dist\_a = random\_state.normal(0.8, 0.05, 1000)

dist\_b = random\_state.normal(0.4, 0.02, 1000)

dist\_c = random\_state.normal(0.6, 0.1, 1000)

# Concatena i dati per simulare un data stream con 2 drifts

stream = np.concatenate((dist\_a, dist\_b, dist\_c))

# Funzione ausiliaria per plottare i dati

def plot\_data(dist\_a, dist\_b, dist\_c, drifts=None):

fig = plt.figure(figsize=(7,3), tight\_layout=True)

gs = gridspec.GridSpec(1, 2, width\_ratios=[3, 1])

ax1, ax2 = plt.subplot(gs[0]), plt.subplot(gs[1])

ax1.grid()

ax1.plot(stream, label='Stream')

ax2.grid(axis='y')

ax2.hist(dist\_a, label=r'$dist\_a$')

ax2.hist(dist\_b, label=r'$dist\_b$')

ax2.hist(dist\_c, label=r'$dist\_c$')

if drifts is not None:

for drift\_detected in drifts:

ax1.axvline(drift\_detected, color='red')

plt.show()

plot\_data(dist\_a, dist\_b, dist\_c)

Immagine che contiene Diagramma, linea, diagramma, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

**Drift detection test**

Useremo il metodo di drift detection ADaptive WINdowing (ADWIN). Ricorda che l’obiettivo è indicare che si è verificato un drift dopo i samples 1000 e 2000 nel synthetic data stream.

from river import drift

drift\_detector = drift.ADWIN()

drifts = []

for i, val in enumerate(stream):

drift\_detector.update(val) # I dati processati un sample alla volta

if drift\_detector.drift\_detected:

# Il drift detector indica dopo ogni sample se c’è un drift nei dati

print(f'Change detected at index {i}')

drifts.append(i)

plot\_data(dist\_a, dist\_b, dist\_c, drifts)

Change detected at index 1023

Change detected at index 2047

Immagine che contiene Diagramma, linea, diagramma, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Vediamo che ADWIN indica con successo la presenza di drift (linee verticali rosse) vicino all’inizio di una nuova distribuzione di dati.

Concludiamo questo esempio con alcune osservazioni riguardo i concept drift detectors e il loro utilizzo:

* In pratica, i drift detectors forniscono ai metodi di stream learning una robustezza contro il concept drift. I drift detectors monitorano il modello di solito attraverso una metrica di performance.
* I drift detectors lavorano su dati *univariate*. Per questo motivo vengono usati per monitorare la performance di un modello e non i dati stessi. Ricorda che il concept drift è definito come un cambiamento nella relazione tra dati e target da apprendere (nel supervised learning).
* I drift detectors definiscono le loro aspettative riguardo i dati in input. È importante conoscere queste aspettative per fornire a un dato drift detector i dati corretti.

**Multi-class classification**

La classification riguarda la previsione di un outcome da una lista fissa di classi. La previsione è una distribuzione di probabilità che assegna una probabilità a ciascun possibile outcome.

Un sample etichettato per classificazione è composto da un insieme di *features* e da una classe. La classe è di solito una stringa o un numero nel caso di *multiclass classification*. Usiamo come esempio il dataset *image segments*.

from river import datasets

dataset = datasets.ImageSegments()

dataset

Image segments classification

Questo dataset contiene features che descrivono segmenti di immagini in 7 classi: brickface, sky, foliage, cement, window, path, e grass.

Name ImageSegments

Task Multi-class classification

Samples 2,310

Features 18

Classes 7

Sparse False

Path /home/runner/work/river/river/river/datasets/segment.csv.zip

Questo dataset è uno *streaming dataset* che può essere iterato.

for x, y in dataset:

pass

Diamo un’occhiata al primo sample:

x, y = next(iter(dataset))

x

{

'region-centroid-col': 218,

'region-centroid-row': 178,

'short-line-density-5': 0.11111111,

'short-line-density-2': 0.0,

'vedge-mean': 0.8333326999999999,

'vegde-sd': 0.54772234,

'hedge-mean': 1.1111094,

'hedge-sd': 0.5443307,

'intensity-mean': 59.629630000000006,

'rawred-mean': 52.44444300000001,

'rawblue-mean': 75.22222,

'rawgreen-mean': 51.22222,

'exred-mean': -21.555555,

'exblue-mean': 46.77778,

'exgreen-mean': -25.222220999999998,

'value-mean': 75.22222,

'saturation-mean': 0.31899637,

'hue-mean': -2.0405545

}

Y

'path'

L’obiettivo di un multiclass classifier è imparare a predire una classe y a partire da un insieme di features X. Proveremo a farlo con un *decision tree*.

from river import tree

model = tree.HoeffdingTreeClassifier()

model.predict\_proba\_one(x)

{}

Il motivo per cui il dizionario di output è vuoto è che il modello non ha ancora visto alcun dato. Non è a conoscenza del dataset. Se questo fosse un **binary classifier**, allora restituirebbe una probabilità del 50% per True e False perché le classi sono implicite. Ma in questo caso stiamo facendo multiclass classification.

Allo stesso modo, il metodo predict\_one inizialmente restituisce None perché il modello non ha ancora visto dati etichettati.

print(model.predict\_one(x))

None

Se aggiorniamo il modello e riproviamo, vediamo che viene assegnata una probabilità del 100% alla classe 'path' perché è l’unica che il modello conosce in quel momento.

model.learn\_one(x, y)

model.predict\_proba\_one(x)

{'path': 1.0}

Questa è una forza dei classificatori online: sono in grado di gestire nuove classi che appaiono nel data stream.

Tipicamente, un modello online fa una previsione e poi apprende una volta che il ground truth si rivela. La previsione e il ground truth possono essere confrontati per misurare la correttezza del modello. Se hai un dataset disponibile, puoi iterarlo, fare una previsione, aggiornare il modello e confrontare l’output del modello con il ground truth. Questo si chiama *progressive validation*.

from river import metrics

model = tree.HoeffdingTreeClassifier()

metric = metrics.ClassificationReport()

for x, y in dataset:

y\_pred = model.predict\_one(x)

model.learn\_one(x, y)

if y\_pred is not None:

metric.update(y, y\_pred)

metric

Precision Recall F1 Support

brickface 77.13% 84.85% 80.81% 330

cement 78.92% 83.94% 81.35% 330

foliage 65.69% 20.30% 31.02% 330

grass 100.00% 96.97% 98.46% 330

path 90.63% 91.19% 90.91% 329

sky 99.08% 98.18% 98.63% 330

window 43.50% 67.88% 53.02% 330

Macro 79.28% 77.62% 76.31%

Micro 77.61% 77.61% 77.61%

Weighted 79.27% 77.61% 76.31%

77.61% accuracy

Questo è un modo comune per valutare un modello online. In realtà, c’è una funzione dedicata evaluate.progressive\_val\_score che lo fa per te.

from river import evaluate

model = tree.HoeffdingTreeClassifier()

metric = metrics.ClassificationReport()

evaluate.progressive\_val\_score(dataset, model, metric)

Precision Recall F1 Support

brickface 77.13% 84.85% 80.81% 330

cement 78.92% 83.94% 81.35% 330

foliage 65.69% 20.30% 31.02% 330

grass 100.00% 96.97% 98.46% 330

path 90.63% 91.19% 90.91% 329

sky 99.08% 98.18% 98.63% 330

window 43.50% 67.88% 53.02% 330

Macro 79.28% 77.62% 76.31%

Micro 77.61% 77.61% 77.61%

Weighted 79.27% 77.61% 76.31%

77.61% accuracy

**Regression**

La regressione riguarda la previsione di un output numerico per un dato sample. Un sample etichettato per regressione è composto da un insieme di features e da un numero. Il numero è solitamente continuo, ma può anche essere discreto. Usiamo come esempio il dataset *Trump approval rating*.

from river import datasets

dataset = datasets.TrumpApproval()

dataset

Donald Trump approval ratings.

Questo dataset è stato ottenuto ristrutturando i dati usati da FiveThirtyEight per analizzare i livelli di approvazione di Donald Trump. Contiene 5 features, che sono indici di approvazione raccolti da 5 agenzie di sondaggi. Il target è l’indice di approvazione derivato dal modello di FiveThirtyEight. L’obiettivo di questo task è verificare se possiamo riprodurre il modello di FiveThirtyEight.

Name TrumpApproval

Task Regression

Samples 1,001

Features 6

Sparse False

Path /home/runner/work/river/river/river/datasets/trump\_approval.csv.gz

Questo dataset è uno streaming dataset che può essere iterato.

for x, y in dataset:

pass

Diamo un’occhiata al primo sample:

x, y = next(iter(dataset))

x

{

'ordinal\_date': 736389,

'gallup': 43.843213,

'ipsos': 46.19925042857143,

'morning\_consult': 48.318749,

'rasmussen': 44.104692,

'you\_gov': 43.636914000000004

}

**Regressor example**

L’obiettivo di un *regression model* è imparare a predire un target numerico y a partire da un insieme di features X. Proveremo a farlo con un *nearest neighbors model*.

from river import neighbors

model = neighbors.KNNRegressor()

model.predict\_one(x)

0.0

Il modello non è stato addestrato su nessun dato e quindi restituisce un valore di default pari a 0.

Il modello può essere addestrato sul sample, aggiornando il suo stato:

model.learn\_one(x, y)

Se proviamo a fare una previsione sullo stesso sample, vediamo che l’output è diverso, perché il modello ha appreso qualcosa:

model.predict\_one(x)

43.7550

Tipicamente, un modello online fa una previsione e poi apprende una volta che il ground truth si riveli. La previsione e il ground truth possono essere confrontati per misurare la correttezza del modello. Se hai un dataset disponibile, puoi iterarlo, fare una previsione, aggiornare il modello e confrontare l’output del modello con il ground truth. Questo si chiama *progressive validation*.

from river import metrics

model = neighbors.KNNRegressor()

metric = metrics.MAE()

for x, y in dataset:

y\_pred = model.predict\_one(x)

model.learn\_one(x, y)

metric.update(y, y\_pred)

metric

MAE: 0.310353

Questo è un metodo comune per valutare un modello online. In realtà, esiste anche una funzione dedicata evaluate.progressive\_val\_score che lo fa per te.

from river import evaluate

model = neighbors.KNNRegressor()

metric = metrics.MAE()

evaluate.progressive\_val\_score(dataset, model, metric)

MAE: 0.310353

**Dal batch all’online/stream**

**Una panoramica veloce del batch learning**

Se ti sei già immerso nel machine learning, allora non dovresti avere difficoltà a usare l’incremental learning. Se invece sei relativamente nuovo nel campo, non preoccuparti! Lo scopo di questo notebook è introdurre nozioni semplici. Inizieremo anche a mostrare come River si inserisce nel contesto e spiegheremo come usarlo.

Il punto fondamentale del machine learning è imparare dai dati. Nel supervised learning vuoi imparare a predire un target dato un set di features. Nel caso di unsupervised learning invece non c’è un target, e l’obiettivo è identificare pattern e trend nelle features. A questo punto la maggior parte delle persone immagina i dati come una tabella: ogni riga è un’osservazione e ogni colonna è una feature. Ed è giusto così. Imparare da dati tabellari fa parte di quello che si chiama batch learning, che significa che tutti i dati sono disponibili in una volta sola per l’algoritmo di apprendimento. Esistono molte librerie per gestire questo regime, una delle più note è scikit-learn di Python.

**Un esempio di batch learning**

Supponiamo di voler predire se una donna ha o meno un tumore al seno. Usiamo il dataset breast cancer disponibile in scikit-learn. Impariamo a mappare un set di features su una decisione binaria usando una logistic regression. Come altri modelli basati su pesi numerici, la logistic regression è sensibile alla scala delle features. Portare ogni feature a media 0 e varianza 1 è considerata buona pratica. Possiamo fare rescaling e fit in maniera elegante con una Pipeline. Per misurare la performance valuteremo la media della ROC AUC score con una cross-validation a 5 fold.

from sklearn import datasets

from sklearn import linear\_model

from sklearn import metrics

from sklearn import model\_selection

from sklearn import pipeline

from sklearn import preprocessing

# Carichiamo i dati

dataset = datasets.load\_breast\_cancer()

X, y = dataset.data, dataset.target

# Definiamo i passaggi del modello

model = pipeline.Pipeline([

('scale', preprocessing.StandardScaler()),

('lin\_reg', linear\_model.LogisticRegression(solver='lbfgs'))

])

# Procedura di cross-validation deterministica

cv = model\_selection.KFold(n\_splits=5, shuffle=True, random\_state=42)

# Calcoliamo lo score

scorer = metrics.make\_scorer(metrics.roc\_auc\_score)

scores = model\_selection.cross\_val\_score(model, X, y, scoring=scorer, cv=cv)

# Mostriamo media e deviazione standard

print(f'ROC AUC: {scores.mean():.3f} (± {scores.std():.3f})')

Output:

ROC AUC: 0.975 (± 0.011)

Batch learning si riduce a:

1. Caricare (e preprocessare) i dati
2. Fittare un modello sui dati
3. Calcolare la performance su dati non visti

Questo è lo standard ed è come molti immaginano una pipeline di machine learning. Tuttavia, ci sono limiti.

* Se un dataset è più grande della tua RAM, il laptop crasha. Si possono usare trucchi (tipi di dato ottimizzati, rappresentazioni sparse), ma non basta con dataset di centinaia di GB: serve hardware speciale.
* Non gestisce bene nuovi dati. Se arrivano dati freschi, il modello deve ri-addestrarsi da zero con vecchi + nuovi dati. Questo è un problema se i dati arrivano continuamente (giornalmente, orariamente o addirittura al secondo). Ad esempio un recommendation engine di e-commerce viene spesso riaddestrato da zero ogni settimana. Con la crescita dell’app, cresce anche il dataset e i tempi di training diventano sempre più lunghi, fino a richiedere upgrade hardware.
* Feature extraction: a volte alcune feature non sono più disponibili al momento dell’addestramento (es. dati sovrascritti nel data warehouse). Questo è frequente nella realtà.

**Introduzione pratica all’incremental learning**

Incremental learning è anche chiamato *online learning* o *stream learning*. L’idea è fittare un modello su uno stream di dati: non hai tutto il dataset, ma osservazioni che arrivano una alla volta.

Esempio: stream del dataset precedente.

for xi, yi in zip(X, y):

# Qui il modello impara

pass

Qui il dataset è già in memoria, ma potremmo streammare da CSV, Kafka, query SQL, ecc. Il problema è che xi è un numpy.ndarray.

xi

array([7.760e+00, 2.454e+01, 4.792e+01, 1.810e+02, 5.263e-02, 4.362e-02,

0.000e+00, 0.000e+00, 1.587e-01, 5.884e-02, 3.857e-01, 1.428e+00,

2.548e+00, 1.915e+01, 7.189e-03, 4.660e-03, 0.000e+00, 0.000e+00,

2.676e-02, 2.783e-03, 9.456e+00, 3.037e+01, 5.916e+01, 2.686e+02,

8.996e-02, 6.444e-02, 0.000e+00, 0.000e+00, 2.871e-01, 7.039e-02])

River invece lavora con dict, che sono più comodi: le feature si accedono per nome e non per posizione.

for xi, yi in zip(X, y):

xi = dict(zip(dataset.feature\_names, xi))

pass

xi

{

'mean radius': 7.76,

'mean texture': 24.54,

'mean perimeter': 47.92,

'mean area': 181.0,

'mean smoothness': 0.05263,

'mean compactness': 0.04362,

'mean concavity': 0.0,

'mean concave points': 0.0,

'mean symmetry': 0.1587,

'mean fractal dimension': 0.05884,

'radius error': 0.3857,

'texture error': 1.428,

'perimeter error': 2.548,

'area error': 19.15,

'smoothness error': 0.007189,

'compactness error': 0.00466,

'concavity error': 0.0,

'concave points error': 0.0,

'symmetry error': 0.02676,

'fractal dimension error': 0.002783,

'worst radius': 9.456,

'worst texture': 30.37,

'worst perimeter': 59.16,

'worst area': 268.6,

'worst smoothness': 0.08996,

'worst compactness': 0.06444,

'worst concavity': 0.0,

'worst concave points': 0.0,

'worst symmetry': 0.2871,

'worst fractal dimension': 0.07039

}

Il modulo stream di River ha anche una funzione pronta chiamata iter\_sklearn\_dataset che possiamo utilizzare:

from river import stream

for xi, yi in stream.iter\_sklearn\_dataset(datasets.load\_breast\_cancer()):

pass

Il semplice fatto che stiamo ricevendo i dati come uno stream significa che non possiamo fare molte cose nello stesso modo in cui le facciamo in un contesto batch. Ad esempio, supponiamo di voler scalare i dati in modo che abbiano media 0 e varianza 1, come abbiamo fatto prima. Per farlo dobbiamo semplicemente sottrarre la media di ogni feature ad ogni valore e poi dividere il risultato per la deviazione standard della feature. Il problema è che non possiamo conoscere i valori della media e della deviazione standard prima di aver attraversato tutti i dati!

Un modo per procedere sarebbe fare un primo passaggio sui dati per calcolare i valori necessari e poi fare lo scaling durante un secondo passaggio. Il problema è che questo vanifica il nostro scopo, che è quello di imparare guardando i dati una sola volta. Anche se può sembrare piuttosto restrittivo, porta benefici significativi sul lungo periodo.

Il modo in cui facciamo feature scaling in **River** implica il calcolo di statistiche progressive (chiamate anche moving statistics). L’idea è usare una struttura dati che stimi la media e si aggiorni ogni volta che riceve un nuovo valore. Lo stesso vale per la varianza (e quindi per la deviazione standard). Ad esempio, se indichiamo con μ\_t la media e con n\_t il conteggio in un dato momento, allora l’aggiornamento della media può essere fatto così:

Immagine che contiene Carattere, testo, calligrafia, bianco

Il contenuto generato dall'IA potrebbe non essere corretto.

Allo stesso modo, la varianza progressiva può essere calcolata così:

Immagine che contiene testo, Carattere, calligrafia, bianco

Il contenuto generato dall'IA potrebbe non essere corretto.

Dove s\_t è una somma progressiva dei quadrati e σ\_t è la varianza progressiva al tempo t. Questo può sembrare un po’ più complicato degli algoritmi batch che si studiano a scuola, ma in realtà è piuttosto elegante. Implementarlo in Python non è difficile. Per esempio, calcoliamo la media e la varianza progressive della variabile *mean area*.

n, mean, sum\_of\_squares, variance = 0, 0, 0, 0

for xi, yi in stream.iter\_sklearn\_dataset(datasets.load\_breast\_cancer()):

n += 1

old\_mean = mean

mean += (xi['mean area'] - mean) / n

sum\_of\_squares += (xi['mean area'] - old\_mean) \* (xi['mean area'] - mean)

variance = sum\_of\_squares / n

print(f'Running mean: {mean:.3f}')

print(f'Running variance: {variance:.3f}')

Output:

Running mean: 654.889

Running variance: 123625.903

Confrontiamo ora con numpy. Ricorda però che numpy richiede accesso a *tutti* i dati.

import numpy as np

i = list(dataset.feature\_names).index('mean area')

print(f'True mean: {np.mean(X[:, i]):.3f}')

print(f'True variance: {np.var(X[:, i]):.3f}')

Output:

True mean: 654.889

True variance: 123625.903

I risultati sembrano essere esattamente gli stessi! La particolarità è che le statistiche progressive non sono molto accurate nelle prime osservazioni. In generale però questo non conta troppo. Alcuni arrivano persino a dire che questa discrepanza è benefica e agisce come una sorta di regularization…

Ora l’idea è che possiamo calcolare le statistiche progressive di ciascuna feature e scalarle man mano che arrivano. Il modo per farlo con River è usare la classe StandardScaler dal modulo preprocessing, così:

from river import preprocessing

scaler = preprocessing.StandardScaler()

for xi, yi in stream.iter\_sklearn\_dataset(datasets.load\_breast\_cancer()):

scaler.learn\_one(xi)

Ora che stiamo scalando i dati, possiamo iniziare a fare del vero machine learning. Andremo a implementare un compito di online linear regression. Poiché tutti i dati non sono disponibili in una volta sola, siamo obbligati a usare quello che si chiama *stochastic gradient descent (SGD*), che è un argomento molto studiato e con molte varianti. L’SGD è comunemente usato per addestrare le reti neurali.

L’idea è che a ogni passo calcoliamo la loss tra la predizione del target e la verità. Poi calcoliamo il gradiente, che è semplicemente l’insieme delle derivate rispetto a ciascun peso della regressione lineare. Una volta ottenuto il gradiente, possiamo aggiornare i pesi spostandoli nella direzione opposta al gradiente. L’entità dello spostamento dipende da un *learning rate*, tipicamente impostato dall’utente. Diversi ottimizzatori hanno modi diversi di gestire l’aggiornamento dei pesi, e alcuni gestiscono il learning rate implicitamente.

L’online linear regression si può fare in River con la classe *LinearRegression* dal modulo linear\_model. In questo caso useremo una logistic regression con SGD, utilizzando l’ottimizzatore SGD dal modulo optim. Durante l’addestramento misureremo lo squared error tra verità e predizioni.

from river import linear\_model

from river import optim

scaler = preprocessing.StandardScaler()

optimizer = optim.SGD(lr=0.01)

log\_reg = linear\_model.LogisticRegression(optimizer)

y\_true = []

y\_pred = []

for xi, yi in stream.iter\_sklearn\_dataset(datasets.load\_breast\_cancer(), shuffle=True, seed=42):

# Scale delle features

scaler.learn\_one(xi)

xi\_scaled = scaler.transform\_one(xi)

# Test sul nuovo campione "non osservato"

yi\_pred = log\_reg.predict\_proba\_one(xi\_scaled)

# Addestriamo il modello con il nuovo campione

log\_reg.learn\_one(xi\_scaled, yi)

# Salviamo la verità e la predizione

y\_true.append(yi)

y\_pred.append(yi\_pred[True])

print(f'ROC AUC: {metrics.roc\_auc\_score(y\_true, y\_pred):.3f}')

Output:

ROC AUC: 0.990

La ROC AUC è significativamente migliore di quella ottenuta con la cross-validation della logistic regression di scikit-learn. Tuttavia, per rendere davvero comparabili i risultati, sarebbe opportuno confrontare con la stessa procedura di cross-validation.

River ha un modulo *compat* che contiene strumenti per renderlo compatibile con altre librerie Python. Poiché stiamo facendo regressione, useremo lo *SKLRegressorWrapper*. Inoltre useremo una Pipeline per incapsulare la logica di StandardScaler e LogisticRegression in un unico oggetto.

from river import compat

from river import compose

# Definiamo una Pipeline, esattamente come prima in sklearn

model = compose.Pipeline(

('scale', preprocessing.StandardScaler()),

('log\_reg', linear\_model.LogisticRegression())

)

# Rendiamo la Pipeline compatibile con sklearn

model = compat.convert\_river\_to\_sklearn(model)

# Calcoliamo gli score di CV usando lo stesso schema e lo stesso scoring

scores = model\_selection.cross\_val\_score(model, X, y, scoring=scorer, cv=cv)

# Mostriamo la media e la deviazione standard

print(f'ROC AUC: {scores.mean():.3f} (± {scores.std():.3f})')

Output:

ROC AUC: 0.964 (± 0.016)

Questa volta la ROC AUC è più bassa, il che è ciò che ci aspettavamo. In effetti l’online learning non è accurato quanto il batch learning. Tuttavia dipende da cosa ti interessa: se vuoi solo predire la prossima osservazione, allora il regime di online learning è migliore. Ecco perché è difficile confrontare i due approcci: sono entrambi adatti a scenari diversi.

**Previsione per il bike-sharing**

In questo tutorial andremo a prevedere il numero di biciclette disponibili in 5 stazioni di bike-sharing della città di Tolosa. Lo faremo costruendo un modello semplice passo dopo passo. Il dataset contiene 182.470 osservazioni. Diamo prima un’occhiata ai dati.

from pprint import pprint

from river import datasets

dataset = datasets.Bikes()

for x, y in dataset:

pprint(x)

print(f'Number of available bikes: {y}')

break

Output:

{'clouds': 75,

'description': 'light rain',

'humidity': 81,

'moment': datetime.datetime(2016, 4, 1, 0, 0, 7),

'pressure': 1017.0,

'station': 'metro-canal-du-midi',

'temperature': 6.54,

'wind': 9.3}

Number of available bikes: 1

Iniziamo usando una semplice linear regression sulle feature numeriche. Possiamo selezionare le feature numeriche ed eliminare le altre usando un *Select*. La linear regression può dare risultati completamente errati se non scaliamo i dati, quindi useremo uno *StandardScaler* per farlo. Valuteremo il modello misurando il *mean absolute error (MAE)*. Infine stamperemo lo score ogni 20.000 osservazioni.

from river import compose

from river import linear\_model

from river import metrics

from river import evaluate

from river import preprocessing

from river import optim

model = compose.Select('clouds', 'humidity', 'pressure', 'temperature', 'wind')

model |= preprocessing.StandardScaler()

model |= linear\_model.LinearRegression(optimizer=optim.SGD(0.001))

metric = metrics.MAE()

evaluate.progressive\_val\_score(dataset, model, metric, print\_every=20\_000)

Output:

[20,000] MAE: 4.912763

[40,000] MAE: 5.333578

[60,000] MAE: 5.330969

[80,000] MAE: 5.392334

[100,000] MAE: 5.423078

[120,000] MAE: 5.541239

[140,000] MAE: 5.613038

[160,000] MAE: 5.622441

[180,000] MAE: 5.567836

[182,470] MAE: 5.563905

MAE: 5.563905

Il modello non sembra funzionare molto bene, ma d’altra parte non abbiamo fornito molte feature.  
Generalmente, una buona idea in questo tipo di problemi è considerare la media dei valori precedenti. Per esempio, per ogni stazione possiamo guardare al numero medio di biciclette per ora. Per farlo dobbiamo prima estrarre l’ora dal campo *moment*. Poi possiamo usare un *TargetAgg* per aggregare i valori del target.

from river import feature\_extraction

from river import stats

def get\_hour(x):

x['hour'] = x['moment'].hour

return x

model = compose.Select('clouds', 'humidity', 'pressure', 'temperature', 'wind')

model += (

get\_hour |

feature\_extraction.TargetAgg(by=['station', 'hour'], how=stats.Mean())

)

model |= preprocessing.StandardScaler()

model |= linear\_model.LinearRegression(optimizer=optim.SGD(0.001))

metric = metrics.MAE()

evaluate.progressive\_val\_score(dataset, model, metric, print\_every=20\_000)

Output:

[20,000] MAE: 3.720766

[40,000] MAE: 3.829739

[60,000] MAE: 3.844905

[80,000] MAE: 3.910137

[100,000] MAE: 3.888553

[120,000] MAE: 3.923644

[140,000] MAE: 3.980882

[160,000] MAE: 3.949972

[180,000] MAE: 3.934489

[182,470] MAE: 3.933442

MAE: 3.933442

Aggiungendo una sola feature, siamo riusciti a ridurre significativamente il MAE. A questo punto potresti pensare che il modello stia diventando leggermente complesso e difficile da capire e testare. Le *Pipeline* hanno il vantaggio di essere concise, ma non sempre facili da fare debug. Per fortuna River offre modi per semplificare il lavoro.

La prima cosa che possiamo fare è visualizzare la pipeline, per capire come i dati scorrono al suo interno.

Model

Immagine che contiene testo, schermata, diagramma, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

Possiamo anche usare il metodo *debug\_one* per vedere cosa succede a una particolare istanza. Alleniamo il modello sulle prime 10.000 osservazioni e poi chiamiamo debug\_one sulla successiva. Per farlo, trasformiamo l’oggetto *Bike* in un generatore Python con la funzione *iter().* Il modo “pythonic” per leggere i primi 10.000 elementi di un generatore è usare *itertools.islice*.

import itertools

model = compose.Select('clouds', 'humidity', 'pressure', 'temperature', 'wind')

model += (

get\_hour |

feature\_extraction.TargetAgg(by=['station', 'hour'], how=stats.Mean())

)

model |= preprocessing.StandardScaler()

model |= linear\_model.LinearRegression()

for x, y in itertools.islice(dataset, 10000):

y\_pred = model.predict\_one(x)

model.learn\_one(x, y)

x, y = next(iter(dataset))

print(model.debug\_one(x))

Output:

0. Input

--------

clouds: 75 (int)

description: light rain (str)

humidity: 81 (int)

moment: 2016-04-01 00:00:07 (datetime)

pressure: 1,017.00000 (float)

station: metro-canal-du-midi (str)

temperature: 6.54000 (float)

wind: 9.30000 (float)

1. Transformer union

--------------------

1.0 Select

----------

clouds: 75 (int)

humidity: 81 (int)

pressure: 1,017.00000 (float)

temperature: 6.54000 (float)

wind: 9.30000 (float)

1.1 get\_hour | y\_mean\_by\_station\_and\_hour

-----------------------------------------

y\_mean\_by\_station\_and\_hour: 4.43243 (float)

clouds: 75 (int)

humidity: 81 (int)

pressure: 1,017.00000 (float)

temperature: 6.54000 (float)

wind: 9.30000 (float)

y\_mean\_by\_station\_and\_hour: 4.43243 (float)

2. StandardScaler

-----------------

clouds: 0.47566 (float)

humidity: 0.42247 (float)

pressure: 1.05314 (float)

temperature: -1.22098 (float)

wind: 2.21104 (float)

y\_mean\_by\_station\_and\_hour: -0.59098 (float)

3. LinearRegression

-------------------

Name Value Weight Contribution

Intercept 1.00000 6.58252 6.58252

pressure 1.05314 3.78529 3.98646

humidity 0.42247 1.44921 0.61225

y\_mean\_by\_station\_and\_hour -0.59098 0.54167 -0.32011

clouds 0.47566 -1.92255 -0.91448

wind 2.21104 -0.77720 -1.71843

temperature -1.22098 2.47030 -3.01619

Prediction: 5.21201

Il metodo *debug\_one* mostra cosa succede a un insieme di feature in input, passo dopo passo.

E ora arriva la parte interessante. Fino ad ora abbiamo usato il metodo *progressive\_val\_score* dal modulo evaluate. Questo metodo predice sequenzialmente l’output di un’osservazione e aggiorna subito dopo il modello. Questo modo di procedere è spesso usato per valutare modelli di online learning. Ma in alcuni casi è l’approccio sbagliato.

Quando valutiamo un modello di machine learning, l’obiettivo è simulare le condizioni di produzione per avere una valutazione affidabile della sua performance. Nel nostro caso, tipicamente vogliamo prevedere il numero di biciclette disponibili in una stazione, diciamo, con 30 minuti di anticipo. Poi, una volta passati i 30 minuti, il numero reale di biciclette disponibili sarà noto e potremo aggiornare il modello usando le feature disponibili 30 minuti prima.

Quello che vogliamo davvero è valutare il modello facendo previsioni con 30 minuti di anticipo e aggiornandolo solo quando i valori reali diventano disponibili. Questo si può fare usando i parametri *moment* e *delay* del metodo *progressive\_val\_score*.

L’idea è che ogni osservazione nello stream dei dati venga mostrata due volte al modello: la prima volta per fare la previsione, e la seconda volta per aggiornare il modello quando il valore reale è rivelato. Il parametro *moment* determina quale variabile usare come timestamp, mentre il parametro *delay* controlla quanto tempo aspettare prima di rivelare i valori reali al modello.

import datetime as dt

evaluate.progressive\_val\_score(

dataset=dataset,

model=model.clone(),

metric=metrics.MAE(),

moment='moment',

delay=dt.timedelta(minutes=30),

print\_every=20\_000

)

Output:

[20,000] MAE: 20.198137

[40,000] MAE: 12.199763

[60,000] MAE: 9.468279

[80,000] MAE: 8.126625

[100,000] MAE: 7.273133

[120,000] MAE: 6.735469

[140,000] MAE: 6.376704

[160,000] MAE: 6.06156

[180,000] MAE: 5.806744

[182,470] MAE: 5.780772

MAE: 5.780772

La performance è un po’ peggiore, come ci si aspettava. In effetti il compito è più difficile: il modello vede il valore reale solo 30 minuti dopo aver fatto la predizione.

La conclusione di questo notebook è che il metodo *progressive\_val\_score* può essere usato per simulare uno scenario di produzione, ed è quindi estremamente prezioso.

**Costruire un semplce modello di nowcasting**

Il *nowcasting* è un caso particolare di *forecasting*. Consiste semplicemente nel predire il prossimo valore in una *time series*.

Useremo i dati internazionali sui passeggeri aerei disponibili qui. Questo *dataset* è incluso in River nel modulo datasets.

from river import datasets

for x, y in datasets.AirlinePassengers():

print(x, y)

break

{'month': datetime.datetime(1949, 1, 1, 0, 0)} 112

I dati sono molto semplici: consistono in una sequenza di mesi e valori che rappresentano il numero totale di passeggeri aerei internazionali per mese.  
Il nostro obiettivo sarà predire il numero di passeggeri del mese successivo a ogni passo.

Nota che, dato che il *dataset* è piccolo (di solito è così per le *time series*), potremmo semplicemente addestrare un modello da zero ogni mese. Tuttavia, per l’esempio, useremo un singolo modello addestrato online. Anche se le performance potrebbero essere più deboli, un modello di *time series* addestrato online ha il vantaggio di scalare facilmente se, ad esempio, hai migliaia di *time series* da gestire.

Partiamo con un modello molto semplice dove l’unica *feature* sarà la data ordinale di ciascun mese. Questo dovrebbe catturare almeno il trend di base.

from river import compose

from river import linear\_model

from river import preprocessing

def get\_ordinal\_date(x):

return {'ordinal\_date': x['month'].toordinal()}

model = compose.Pipeline(

('ordinal\_date', compose.FuncTransformer(get\_ordinal\_date)),

('scale', preprocessing.StandardScaler()),

('lin\_reg', linear\_model.LinearRegression())

)

Scriviamo una funzione per valutare il modello. Scorrerà ogni osservazione del *dataset* aggiornando il modello passo dopo passo. Le predizioni precedenti saranno salvate insieme ai valori reali e poi tracciate in un grafico.

from river import metrics

from river import utils

import matplotlib.pyplot as plt

def evaluate\_model(model):

metric = utils.Rolling(metrics.MAE(), 12)

dates = []

y\_trues = []

y\_preds = []

for x, y in datasets.AirlinePassengers():

# Ottieni la predizione precedente e aggiorna il modello in un unico passaggio

y\_pred = model.predict\_one(x)

model.learn\_one(x, y)

# Aggiorna la metrica di errore

metric.update(y, y\_pred)

# Salva i valori reali e le predizioni

dates.append(x['month'])

y\_trues.append(y)

y\_preds.append(y\_pred)

# Disegna i risultati

fig, ax = plt.subplots(figsize=(10, 6))

ax.grid(alpha=0.75)

ax.plot(dates, y\_trues, lw=3, color='#2ecc71', alpha=0.8, label='Ground truth')

ax.plot(dates, y\_preds, lw=3, color='#e74c3c', alpha=0.8, label='Prediction')

ax.legend()

ax.set\_title(metric)

Valutiamo il nostro primo modello:

evaluate\_model(model)

Immagine che contiene testo, Diagramma, linea, diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

Il modello ha catturato un trend ma non quello corretto. Infatti assume che il trend sia lineare, mentre visivamente vediamo che la crescita dei dati accelera col tempo. In altre parole, la seconda derivata della serie è positiva.

Questo è un problema noto nel *time series forecasting*. Ci sono molti modi per affrontarlo (ad esempio una trasformazione Box-Cox). Qui invece linearly *detrend* la serie usando un TargetStandardScaler.

from river import stats

model = compose.Pipeline(

('ordinal\_date', compose.FuncTransformer(get\_ordinal\_date)),

('scale', preprocessing.StandardScaler()),

('lin\_reg', linear\_model.LinearRegression(intercept\_lr=0)),

)

model = preprocessing.TargetStandardScaler(regressor=model)

evaluate\_model(model)

Immagine che contiene Diagramma, linea, testo, diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

Ora proviamo a catturare la stagionalità mensile con un *one-hot encoding* del nome del mese.

import calendar

def get\_month(x):

return {

calendar.month\_name[month]: month == x['month'].month

for month in range(1, 13)

}

model = compose.Pipeline(

('features', compose.TransformerUnion(

('ordinal\_date', compose.FuncTransformer(get\_ordinal\_date)),

('month', compose.FuncTransformer(get\_month)),

)),

('scale', preprocessing.StandardScaler()),

('lin\_reg', linear\_model.LinearRegression(intercept\_lr=0))

)

model = preprocessing.TargetStandardScaler(regressor=model)

evaluate\_model(model)

Immagine che contiene Diagramma, linea, testo, diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

Ottimo risultato. Possiamo anche guardare i *weights* della regressione lineare per capire l’importanza delle *feature*.

model.regressor['lin\_reg'].weights

Output:

{

'January': -0.1380,

'February': -0.1871,

'March': -0.0264,

'April': -0.0350,

'May': -0.0136,

'June': 0.1619,

'July': 0.3199,

'August': 0.2810,

'September': 0.0383,

'October': -0.1165,

'November': -0.2663,

'December': -0.1539,

'ordinal\_date': 1.0234

}

Come atteso, Luglio e Agosto hanno i pesi maggiori perché sono mesi di vacanze, mentre Dicembre ha un peso basso perché è un mese di festività dove di solito si viaggia meno.

Problema: il modello riconosce i mesi importanti, ma non che la loro importanza cresce in modo moltiplicativo negli anni. In pratica il modello è troppo “timido”. Possiamo fixarlo aumentando il *learning rate* dell’ottimizzatore della LinearRegression.

from river import optim

model = compose.Pipeline(

('features', compose.TransformerUnion(

('ordinal\_date', compose.FuncTransformer(get\_ordinal\_date)),

('month', compose.FuncTransformer(get\_month)),

)),

('scale', preprocessing.StandardScaler()),

('lin\_reg', linear\_model.LinearRegression(

intercept\_lr=0,

optimizer=optim.SGD(0.03)

))

)

model = preprocessing.TargetStandardScaler(regressor=model)

evaluate\_model(model)

Immagine che contiene testo, Diagramma, linea, diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

Questo inizia a sembrare buono! Naturalmente in produzione noi regoleremmo il *learning rate*, idealmente in tempo reale.

Prima di finire, introdurremo un trucchetto interessante di *feature extraction* basato sui *radial basis function kernels*. Il *one-hot encoding* che abbiamo fatto sul mese è una buona idea, ma se ci pensi è un po’ rigido. Infatti il valore di ciascuna *feature* sarà 0 o 1, a seconda del mese di ciascuna osservazione. In pratica stiamo dicendo che il mese di Settembre è distante dal mese di Agosto tanto quanto lo è dal mese di Marzo. Ovviamente questo non è vero, e sarebbe bello se le nostre *features* lo riflettessero.

Per farlo possiamo semplicemente calcolare la distanza tra il mese di ciascuna osservazione e tutti i mesi del calendario. Invece di calcolare la distanza linearmente, useremo un cosiddetto *Gaussian radial basis function kernel*. È un nome un po’ complicato, ma per noi si riduce a una formula semplice, che è:

Immagine che contiene Carattere, bianco, calligrafia, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

Intuitivamente questo calcola una similarità tra due mesi — denotati da i e j — che diminuisce quanto più sono lontani tra loro. Il parametro può essere visto come un iperparametro che può essere regolato — nello snippet seguente lo ignoreremo semplicemente. La cosa da notare è che questo produce predizioni più fluide rispetto a quando si usa uno schema *one-hot encoding*, proprietà che spesso è desiderabile. Puoi anche vedere questo trucchetto in azione in questa bella presentazione.

import math

def get\_month\_distances(x):

return {

calendar.month\_name[month]: math.exp(-(x['month'].month - month) \*\* 2)

for month in range(1, 13)

}

model = compose.Pipeline(

('features', compose.TransformerUnion(

('ordinal\_date', compose.FuncTransformer(get\_ordinal\_date)),

('month\_distances', compose.FuncTransformer(get\_month\_distances)),

)),

('scale', preprocessing.StandardScaler()),

('lin\_reg', linear\_model.LinearRegression(

intercept\_lr=0,

optimizer=optim.SGD(0.03)

))

)

model = preprocessing.TargetStandardScaler(regressor=model)

evaluate\_model(model)

Immagine che contiene testo, Diagramma, linea, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

Siamo riusciti a ottenere una curva predittiva dall’aspetto buono con un modello ragionevolmente semplice. In più, il nostro modello ha il vantaggio di essere interpretabile e facile per il *debug*.  
Ci sono sicuramente altri aspetti da spremere (ad esempio regolare gli iperparametri, usare un modello *ensemble*, ecc.) ma lo lasciamo come esercizio al lettore.

Come tocco finale riscriveremo la nostra *pipeline* usando l’operatore |, che viene chiamato “pipe”.

extract\_features = compose.TransformerUnion(get\_ordinal\_date, get\_month\_distances)

scale = preprocessing.StandardScaler()

learn = linear\_model.LinearRegression(

intercept\_lr=0,

optimizer=optim.SGD(0.03)

)

model = extract\_features | scale | learn

model = preprocessing.TargetStandardScaler(regressor=model)

evaluate\_model(model)

Immagine che contiene testo, Diagramma, schermata, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

model

Immagine che contiene testo, schermata, diagramma, Rettangolo

Il contenuto generato dall'IA potrebbe non essere corretto.